

lib/main/ordered-list-nat.ath

```

1
2 # Properties of ordered lists of natural numbers
3
4 load "nat-less.ath"
5 load "list-of.ath"
6
7 #.....
8 # <=L: is a natural number less than or equal to the first element of
9 # a list of natural numbers (true if the list is empty).
10
11 extend-module List {
12 open N
13
14 declare <=L: [N (List N)] -> Boolean
15
16 module <=L {
17 assert empty := (forall x . x <=L nil)
18 assert nonempty :=
19   (forall x y L . x <=L (y :: L) <==> x <= y)
20
21 define left-transitive :=
22   (forall L x y . x <= y & y <=L L ==> x <=L L)
23 define before-all-implies-before-first :=
24   (forall L x . (forall y . y in L ==> x <= y) ==> x <=L L)
25 define append :=
26   (forall L M x . x <=L L & x <=L M ==> x <=L (L join M))
27
28 datatype-cases left-transitive {
29 nil =>
30   pick-any x y
31     assume (x <= y & y <=L nil)
32       (!chain-> [true ==> (x <=L nil) [empty]])
33 | (z :: M) =>
34   pick-any x y
35     assume (x <= y & y <=L (z :: M))
36     conclude (x <=L (z :: M))
37       (!chain->
38         [(x <= y & y <=L (z :: M))
39          ==> (x <= y & y <= z) [nonempty]
40          ==> (x <= z) [Less=.transitive]
41          ==> (x <=L (z :: M)) [nonempty]])
42 }
43
44 datatype-cases before-all-implies-before-first {
45 nil =>
46   pick-any x
47     assume (forall y . y in nil ==> x <= y)
48     conclude (x <=L nil)
49     (!chain-> [true ==> (x <=L nil) [empty]])
50 | (z:N :: L) =>
51   pick-any x
52     assume i := (forall y . y in (z :: L) ==> x <= y)
53     conclude (x <=L (z :: L))
54     (!chain-> [(z = z)
55              ==> (z = z | z in L) [alternate]
56              ==> (z in (z :: L)) [in.nonempty]
57              ==> (x <= z) [i]
58              ==> (x <=L (z :: L)) [nonempty]])
59 }
60
61 datatype-cases append {
62 nil =>
63   pick-any M x
64     (!chain [(x <=L nil & x <=L M)
65            ==> (x <=L M) [right-and]
66            ==> (x <=L (nil join M)) [join.left-empty]])
67 | (u :: N) =>
68   pick-any M x

```

```

69     assume (x <=L (u :: N) & (x <=L M))
70     (!chain->
71     [(x <=L (u :: N))
72     ==> (x <= u) [nonempty]
73     ==> (x <=L (u :: (N join M))) [nonempty]
74     ==> (x <=L ((u :: N) join M)) [join.left-nonempty]])
75 }
76 } # <=L
77
78 #.....
79 # List.ordered: are the natural numbers in a list in order?
80
81 declare ordered: [(List N)] -> Boolean
82
83 module ordered {
84 assert empty := (ordered nil)
85 assert nonempty :=
86   (forall L x . ordered (x :: L) <==> x <=L L & ordered L)
87
88 define head :=
89   (forall L x . ordered (x :: L) ==> x <=L L)
90 define tail :=
91   (forall L x . ordered (x :: L) ==> ordered L)
92
93 conclude head
94   pick-any L x
95     (!chain
96     [(ordered (x :: L))
97     ==> (x <=L L & ordered L) [nonempty]
98     ==> (x <=L L) [left-and]])
99
100 conclude tail
101   pick-any L x
102     (!chain
103     [(ordered (x :: L))
104     ==> (x <=L L & ordered L) [nonempty]
105     ==> (ordered L) [right-and]])
106
107 define first-to-rest-relation :=
108   (forall L x y . ordered (x :: L) & y in L ==> x <= y)
109 define cons :=
110   (forall L x . ordered L & (forall y . y in L ==> x <= y)
111   ==> ordered (x :: L))
112 define append :=
113   (forall L M . ordered L & ordered M &
114   (forall x y . x in L & y in M ==> x <= y)
115   ==> ordered (L join M))
116
117 by-induction first-to-rest-relation {
118   nil =>
119     pick-any x:N y:N
120     assume i := ((ordered (x :: nil)) & y in nil)
121     let {not-in := (!chain->
122     [true ==> (~ y in nil) [in.empty]])}
123     (!from-complements (x <= y) (y in nil) not-in)
124 | (z:N :: M:(List N)) =>
125   let {ind-hyp := (forall ?x ?y .
126   ordered (?x :: M) & ?y in M ==> ?x <= ?y)}
127 conclude (forall ?x ?y .
128   ordered (?x :: (z :: M)) & ?y in (z :: M)
129   ==> ?x <= ?y)
130   pick-any x:N y:N
131     assume ((ordered (x :: (z :: M))) & y in (z :: M))
132     let {p0 :=
133     (!chain->
134     [(ordered (x :: (z :: M)))
135     ==> (x <=L (z :: M) & ordered (z :: M))
136     [nonempty]
137     ==> (x <=L (z :: M) & z <=L M & ordered M)
138     [nonempty]]}

```

```

139     ==> (x <= z & z <=L M & ordered M)
140         [<=L.nonempty]);
141   p1 :=
142     (!chain-> [p0 ==> (ordered M) [prop-taut]]);
143   p2 :=
144     (!chain->
145       [p0 ==> (x <= z & z <=L M) [prop-taut]
146         ==> (x <=L M) [<=L.left-transitive]
147         ==> (x <=L M & ordered M) [augment]
148         ==> (ordered (x :: M)) [nonempty]]);
149   p3 := (!chain->
150     [(y in (z :: M))
151       ==> (y = z | y in M) [in.nonempty]])}
152   (!cases (y = z | y in M)
153     assume (y = z)
154     (!chain-> [p0 ==> (x <= z) [left-and]
155       ==> (x <= y) [(y = z)]]])
156     (!chain [(y in M)
157       ==> (p2 & y in M) [augment]
158       ==> (x <= y) [ind-hyp]]))
159 }
160
161 conclude cons
162 pick-any L x
163   let {p := (forall ?y . ?y in L ==> x <= ?y)}
164   assume (ordered L & p)
165   (!chain->
166     [p ==> (x <=L L) [<=L.before-all-implies-before-first]
167       ==> (x <=L L & ordered L) [augment]
168       ==> (ordered (x :: L)) [nonempty]])
169
170 by-induction append {
171   nil:(List N) =>
172     conclude (forall ?R .
173       ordered nil & ordered ?R &
174       (forall ?x ?y . ?x in nil & ?y in ?R ==> ?x <= ?y)
175       ==> (ordered (nil join ?R)))
176     pick-any R
177       assume ((ordered nil) & (ordered R) &
178         (forall ?x ?y . ?x in nil & ?y in R ==> ?x <= ?y))
179       (!chain->
180         [(ordered R)
181           ==> (ordered (nil join R)) [join.left-empty]])
182   | (z :: L:(List N)) =>
183     let {ind-hyp :=
184       (forall ?R .
185         ordered L & ordered ?R &
186         (forall ?x ?y . ?x in L & ?y in ?R ==> ?x <= ?y)
187         ==> (ordered (L join ?R)))}
188     conclude
189       (forall ?R .
190         ordered (z :: L) & ordered ?R &
191         (forall ?x ?y . ?x in (z :: L) & ?y in ?R ==> ?x <= ?y)
192         ==> (ordered ((z :: L) join ?R)))
193     pick-any R:(List N)
194       let {A1 := (ordered (z :: L));
195         A2 := (ordered R);
196         A3 := (forall ?x ?y .
197           ?x in (z :: L) & ?y in R ==> ?x <= ?y)}
197       assume (A1 & A2 & A3)
198       let {C1 := (!chain->
199         [(ordered (z :: L)) ==> (ordered L) [tail]]);
200         C2 := conclude
201           (forall ?x ?y . ?x in L & ?y in R ==> ?x <= ?y)
202           pick-any x:N y:N
203             assume D := (x in L & y in R)
204             (!chain->
205               [D ==> (x in (z :: L) & y in R) [in.tail]
206                 ==> (x <= y) [A3]]);
207         C3 := conclude (ordered (L join R))
208

```

```

209      (!chain->
210        [C1 ==> (C1 & (ordered R) & C2) [augment]
211          ==> (ordered (L join R))    [ind-hyp]]);
212      C4 := conclude (z <=L R)
213      (!two-cases
214        assume (R = nil)
215        (!chain-> [true ==> (z <=L nil)  [<=L.empty]
216                  ==> (z <=L R)      [(R = nil)]])
217        assume (R /= nil)
218        let {D1 :=
219          conclude (z in (z :: L))
220          (!chain->
221            [(z = z)
222             ==> (z = z | z in L) [alternate]
223              ==> (z in (z :: L)) [in.nonempty]]);
224          D2 := (exists ?u ?M . R = (?u :: ?M));
225          D3 := conclude D2
226          (!chain->
227            [true
228             ==> (R = nil | D2)
229               [(datatype-axioms "List")]
230              ==> ((R /= nil) & (R = nil | D2))
231                 [augment]
232                 ==> D2                               [prop-taut]]])
233          pick-witnesses u M for D3
234          (!chain->
235            [true
236             ==> (u in (u :: M)) [in.head]
237              ==> (u in R)      [(R = (u :: M))]
238              ==> (z in (z :: L) & u in R) [augment]
239              ==> (z <= u)          [A3]
240              ==> (z <=L (u :: M)) [<=L.nonempty]
241              ==> (z <=L R)        [(R = (u :: M))]])
242      conclude (ordered ((z :: L) join R))
243      (!chain->
244        [(ordered (z :: L))
245         ==> ((z <=L L) & ordered L) [nonempty]
246          ==> (z <=L L)             [left-and]
247          ==> ((z <=L L) & C4)      [augment]
248          ==> (z <=L (L join R))    [<=L.append]
249          ==> (z <=L (L join R) & C3) [augment]
250          ==> (ordered (z :: (L join R))) [nonempty]
251          ==> (ordered ((z :: L) join R)) [join.left-nonempty]
252        ])
253    }
254 } # ordered
255 } # List

```